HAL
archives-ouvertes.fr

Introduction aux simulations numériques.

Michel Caffarel

▶ **To cite this version:**

Michel Caffarel. Introduction aux simulations numériques.. DEA. Cours enseigné en 1995 et 1996 au DEA "Champs, Particules, Matières" (Paris 6, 7 et 11) (38 pages)., 2006. <cel-00092932>

HAL Id: cel-00092932

https://cel.archives-ouvertes.fr/cel-00092932

Submitted on 12 Sep 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Introduction aux simulations numériques*

Michel Caffarel
Laboratoire de Chimie Théorique
Tour 22-23, 1er étage, 4 place Jussieu
75252 Paris Cedex 05
e-mail:mc@lct.jussieu.fr
Avril 1996

Sommaire

I Introduction **2**

II Aspects pratiques des simulations **3**
A Organisation physique d'un ordinateur 3
1 Représentation des nombres 3
2 Vitesse finie du processeur 4
3 Capacité de stockage 4
4 Temps d'accès finis à l'information en mémoire 4
B Quelques éléments de Fortran 4
C Quelques éléments d'Unix. 8
1 Système de fichiers 9
2 Processus 10
D L'éditeur de texte "vi" 10
E Quelques conseils de programmation 11
F Aspects finis de l'ordinateur 13
1 Vitesse finie 13
2 Précision finie 14
3 Temps d'accès finis à la mémoire centrale et à la mémoire dure 15

III Dynamique moléculaire. Chaos classique **16**
A Généralités 16
1 Equations différentielles 16
2 Dynamique moléculaire 17
3 Systèmes Hamiltoniens conservatifs et chaos 18
B Apparition du chaos dans un système d'oscillateurs couplés 19

---

*Document écrit à l'occasion d'un cours donné dans le cadre du DEA "Champs Particules Matières"

IV Méthodes de Monte Carlo. Transition de phase dans le modèle d'Ising à 2 D **21**
A Méthodes de Monte Carlo 21
B Algorithme de Metropolis 22
C Implémentation pratique 24
D Simulation du modèle d'Ising à 2 dimensions 26

V Au voisinage de la transition de phase: l'algorithme de Swendsen-Wang **28**
A Algorithme de Swendsen-Wang 28
B Simulation du modèle d'Ising au voisinage de la transition 30

VI Méthodes de diagonalisations exactes et méthodes de projection. L'oscillateur anharmonique quantique **31**
A Théorie 31
B Oscillateur anharmonique quantique 32
1 Diagonalisation exacte 33
2 Méthodes de projection (méthode des puissances, méthode de Lanczòs) 33

VII Modèle d'Heisenberg antiferromagnétique 2D **36**

I. INTRODUCTION

Le but de ce cours est d'illustrer à l'aide de quelques exemples simples comment un ordinateur peut être utilisé pour étudier un problème physique complexe n'admettant ni solution analytique, ni approximation contrôlée satisfaisante. Les problèmes complexes sont évidemment nombreux en physique, particulièrement dans le domaine de la matière condensée où existe une grande diversité de comportements et de structures. D'un point de vue technique, ce sont le plus souvent des problèmes pour lesquels les méthodes perturbatives (perturbation autour d'une solution connue analytiquement) ne fonctionnent pas et/ou des problèmes correspondant à un régime hautement non-linéaire. Dans les cas les moins difficiles, il s'agit simplement de considérer un grand nombre de degrés de liberté en interaction.

Ce cours n'est ni un cours d'informatique, ni un cours d'analyse numérique. Il existe de nombreux ouvrages sur ces aspects-là et on peut s'y reporter seul avec profit. L'objet de ce cours est plutôt d'illustrer comment des simulations simples à mettre en oeuvre peuvent conduire à des résultats difficiles à obtenir autrement. On insistera sur les contraintes fondamentales qui conditionnent le choix de la méthode à utiliser pour un problème spécifique. Un point important est de savoir évaluer la dépendance de l'effort numérique d'une méthode donnée en fonction du paramètre critique de la simulation. Ici, le paramètre critique sera défini comme le paramètre conditionnant la quantité de mémoire et le temps d'exécution nécessaire pour effectuer la simulation (une valeur infinie du paramètre correspondant en général à une simulation exacte). Concrètement, ce paramètre peut être un nombre de particules, un nombre de pas élémentaires de dynamique moléculaire, etc... Il est évidemment important d'utiliser la méthode optimale pour un problème donné. Choisir la mauvaise méthode conduit le plus souvent à ne pas pouvoir en pratique effectuer la simulation (quantité de mémoire requise et/ou temps d'exécution dépassant les limites physiques

de l'ordinateur). En revanche, une fois la bonne méthode choisie, il est souvent possible d'effectuer des simulations honorables sans beaucoup d'efforts. Il faut souligner que les simulations correspondant à ce qu'on appelle "l'état de l'art" (state-of-the-art) ne sont généralement que des simulations dont les algorithmes ont été poussés à leurs limites ultimes, et ne représentent malheureusement bien souvent qu'un travail fastidieux d'analyse numérique n'apportant pas grand chose de plus du point de vue de la connaisance physique du problème. Pour donner un exemple précis, écrire un programme qui permet de simuler un modèle de Hubbard pour un réseau de 12 sites (le modèle de Hubbard est un des modèles fétiches de la communauté des fermions fortement corrélés) peut se faire en quelques jours, atteindre les 16 sites nécessite certainement quelques longues semaines (mois?) de travail, la limite thermodynamique correspondant évidemment à un nombre de sites de l'ordre de 10^{23}...

Un autre point important est de bien prendre en compte les processus physiques sous-jacents lors de la simulation. Ce principe est à la base de toute simulation efficace. Chaque problème est cependant très spécifique de ce point de vue, et seule la mise en oeuvre de cette idée sur des problèmes concrets permet de s'en convaincre. Le cas du modèle d'Ising traité avec l'algorithme de Swendsen-Wang en est une très bonne illustration (voir Section V).

II. ASPECTS PRATIQUES DES SIMULATIONS

A. Organisation physique d'un ordinateur

En fait, il est important de souligner qu'il n'est pas nécessaire de connaître grand chose sur l'organisation physique des ordinateurs pour les utiliser de manière satisfaisante (n'oublions pas que nous ne visons pas la simulation ultime). Le seul point très important à réaliser est ce qu'on peut appeler les aspects "finis" de l'ordinateur. Un ordinateur est composé d'un processeur qui travaille à une vitesse *finie* à l'aide d'une représentation *finie* des objets qu'il manipule (entiers, réels, etc...). Il est associé à une mémoire centrale qui a une capacité de stockage *finie* et à laquelle il accède en temps *fini*. Un processeur peut aussi accéder à un disque dur (mémoire dure), avec évidemment là aussi une capacité et un temps d'accès finis (bien supérieurs d'ailleurs). Ces aspects étant très importants, ils seront illustrés, à titre d'exercices, sur des exemples concrets dans la section II.F.

1. Représentation des nombres

On appelle généralement "mot" l'entité de base manipulé par le processeur. Pour la plupart des stations de travail, le mot est composé de 4 octets, c'est à dire 32 bits (le bit est l'élément élémentaire ultime et il peut prendre que les valeurs 0 ou 1). Les entiers sont généralement par défaut représentés par un mot et les réels également. Dans ce cas, on parle de calculs en réels simple précision. Un calcul effectué à l'aide d'entiers uniquement est un calcul exact. En revanche, dès que des réels sont utilisés, une erreur dite de précision finie est introduite. Cette erreur peut toujours être réduite en représentant les réels à l'aide de plusieurs mots. Par exemple, utiliser 2 mots s'appelle travailler en double precision, 4 mots en quadruple, etc... On évite généralement d'utiliser des précisions trop grandes parce que

les temps de calcul augmentent de manière importante. En fait, la double précision sur huit octets sur des machines destinées au calcul numérique est très souvent "cablée", c'est à dire que le processeur est physiquement préparé à travailler en double précision et les temps de réponse ne sont pas significativement plus grands qu'en simple précision. Au-delà, il s'agit d'une solution "logicielle", c'est à dire qu'un programme a été écrit pour juxtaposer des mots ensemble. Cette solution est donc beaucoup plus lente. Il est important de souligner qu'on peut toujours soi-même juxtaposer autant de mots qu'on veut pour représenter un réel et atteindre ainsi une précision arbitraire (notons que ceci est dejà fait pour vous dans les programmes de calcul symbolique comme Maple ou Mathematica). Ceci peut être très utile dans des applications où une très grande précision est essentielle ou pour vérifier la stabilité numérique d'un algorithme. Ces notions sont illustrées dans les exercices présentés dans la section II.F.

2. Vitesse finie du processeur

Les machines modernes opérent avec une puissance qui leur permet d'effectuer de l'ordre de quelques millions de multiplications réelles à la seconde.

3. Capacité de stockage

Typiquement, sur les stations de travail la mémoire centrale peut stocker quelques dizaines de millions de mots. Les plus grandes mémoires peuvent atteindre quelques milliards de mots.

La capacité de stockage sur disque dur est virtuellement infinie (on peut juxtaposer les disques). Le vrai problème n'est pas la capacité des disques mais le temps d'accès à l'information stockée, voir discussion qui suit.

4. Temps d'accès finis à l'information en mémoire

Une caractéristique très importante des mémoires est le temps qu'il faut pour accéder à un mot stocké. Les temps d'accès à la mémoire centrale sont beaucoup plus courts que ceux associés à la mémoire dure. Dans l'exercice de la section II.F.3 on évaluera très grossièrement cette différence. Un facteur cent est un ordre de grandeur typique.

B. Quelques éléments de Fortran

Le Fortran est le langage de programmation de la communauté scientifique internationale. Même si ce n'est pas, et de beaucoup, le meilleur langage de programmation, c'est celui que vous retrouverez partout dans les laboratoires de recherche, dans les banques de programmes, etc... Il est donc indispensable de l'apprendre. C'est un langage très simple et quelques heures avec un livre quelconque sur le sujet suffiront à rendre expert n'importe

quelle personne motivée. Cet effort préliminaire est évidemment indispensable si vous voulez profiter dans les meilleures conditions de ce cours. Je présente maintenant un programme fictif (surtout ne chercher aucune signification à la tâche accomplie) utilisant quelques unes des principales conventions et structures du Fortran. J'insère dans le texte les commentaires correspondants.

```
      program ExEmPle
```

Le texte principal d'un programme Fortran (excepté pour les signes de commentaires, les étiquettes et les symboles pour indiquer les lignes suites) commence à la 7ième colonne et finit à la 72ième colonne. Méfiez vous de ne pas dépasser cette 72ième colonne.

ExEmPle est le nom du programme principal. Notez que le Fortran ne distingue pas entre les caractères écrits en minuscule ou en majuscule. Les mots: ExEmPle, Exemple, exemple, etc...sont donc équivalents en Fortran.

```
* zone type des variables
```

Ceci est une ligne de commentaires qui n'est pas prise en compte. C'est le premier caractère de la ligne qui définit une ligne commentaire. On peut utiliser indifféremment les caractères * ou c.

```
      integer term1,term2
      real icase,rapid
      double precision pi,deuxpi
      logical logic
```

Déclaration des types de quelques variables. En Fortran, par défaut, tous les mots dont le nom commence par une lettre $\in$ (a-h) sont considérés comme réels, les autres dans l'intervalle (i-n) sont entiers. Toute déclaration explicite du type d'un mot annule cette convention pour le mot. Noter l'existence de la déclaration "implicit none" qui élimine cette convention. Si vous utilisez cette convention, vous devez déclarer le type de toutes les variables ou constantes utilisées dans le programme.

```
C zone des parameters
      parameter(nmax=100,pi=3.1415926535898,deuxpi=2.d0*pi)
```

Déclaration permettant de fixer quelques valeurs constantes. Particulièrement intéressante pour définir les dimensions des tableaux.

```
c* déclaration des tableaux
      dimension tab1(nmax),tab2(0:nmax)
```

Déclaration des tableaux. tab1(nmax) signifie que vous réservez en mémoire centrale nmax éléments de mémoire: tab1(1),tab1(2),...,tab1(nmax=100). Plus généralement, vous pouvez écrire tab2(i:j) pour: tab2(i),tab2(i+1),...,tab2(j), i,j entiers positifs, nuls ou négatifs.

```
** zone communes
      common/c1/term1,term2
```

Définition d'une zone de mémoire centrale commune à plusieurs sous-programmes. c1 est le nom de la zone commune. term1,term2 sont deux entiers (ils ont été déclarés explicitement comme tels au début de ce programme) qui pourront être utilisés dans les sous-programmes où la zone commune aété introduite. Il est impératif de conserver l'ordre, le type (il faut absolument redéfinir le type dans les sous-programmes) et le nom des variables, constantes ou tableaux passés par common.

```
      open(1,file='resultats')
      rewind1
```

Open: Permet d'ouvrir un fichier dont le nom est écrit entre '...'. Le fichier est associé à l'unité logique numéro 1. Une fois ouvert on pourra écrire ou lire des données sur ce fichier par des ordres du type: read(1,*) ou write(1,*). On peut ouvrir autant de fichiers qu'on veut avec des numéros différents. Rewind 1 est un ordre permettant de se positionner en début de fichier dont l'unité logique est 1. Chaque fois qu'un élément est lu ou écrit dans le fichier un curseur de positionnement est incrémenté.

```
      term1=10
      term2=20
      print*,'Entrez la valeur de n'
      read*,n
```

print*,'...': ordre d'écriture le plus simple sur l'écran. "*" signifie l'unité 6 (ou écran) par défaut. Mettre n'importe quel texte entre '...'
read*,n: ordre de lecture le plus simple sur l'écran. "*" signifie l'unité 5 (ou écran) par défaut.

```
      do i=1,n
        tab1(i)=sqrt(float(i))
      enddo
```

Les structures de boucle jouent un rôle très important. Forme plus générale: "do i=idebut,ifin,increment" où l'incrément est un entier positif ou négatif. Par défaut l'incrément est 1.

sqrt ou float: fonctions intrinsèques prédéfinies et chargées lors de l'étape de "load" (chargement). Il existe de nombreuses fonctions intrinsèques prédéfinies: sqrt,cos,sin,exp,float,alog,etc... et leurs versions en double ou même quadruple ("extended") précision.

```
      print*,'Entrez la valeur de logic'
      read*,logic
```

```
if(logic.eq..true.)then
  do i=0,n-1,2
    tab2(i)=exp(tab1(i+1))
  enddo
else
  do i=0,n-1,2
    tab2(i+1)=expression(i)
  enddo
endif
```

Noter la structure "if(proposition logique)then-else-endif" d'utilisation très commune.

"expression(i)" n'est pas ici un tableau mais une fonction définie par l'utilisateur. Il s'agit d'un cas particulier de sous-programme pour lequel l'expression calculée se réduit à un seul nombre. C'est l'équivalent utilisateur des fonctions intrinsèques prédéfinies vues précédemment. On peut utiliser plusieurs arguments.

```
call prog1(n,tab1,tab2,res)
```

Structure générale pour l'appel à un sous-programme. Vérifier avec beaucoup d'attention que le nombre et le type des variables passés par argument correspondent exactement dans l'ordre call et l'ordre subroutine. C'est une source très importante d'erreur.

```
do i=1,n
  write(1,5f10.5)res(i)
enddo
```

Ecriture sur le fichier 1 (résultat) du tableau res. Notez le format qui n'est pas libre (le format libre par défaut est obtenu en remplacant 5f10.5 par un symbole "*"). Ce format est dit "format fixe": 10 caractères pour représenter un nombre flottant avec 5 chiffres après la virgule, 4 caractères pour le signe et les chiffres avant la virgule, un caractère pour le signe "." (Exemple: 100.12345 1234.6789 sont deux chiffres écrits avec ce format). 5 signifiant qu'il y a 5 nombres écrits avec ce format par ligne. Il existe de nombreux formats (voir livres).

```
close(1)
end
```

Ordre de fermeture du fichier correspondant à l'unité logique 1, puis fin du programme principal.

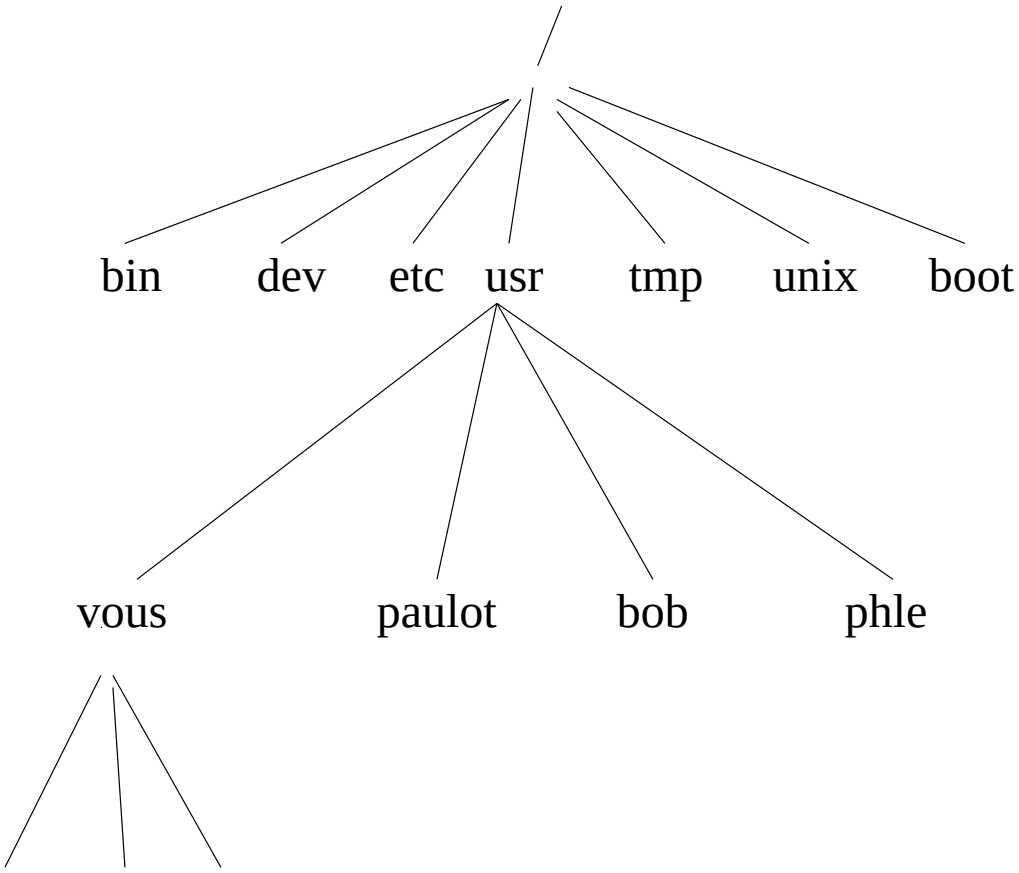
```
function expression(k)
sum=0.
do i=1,k
  sum=sum+float(i)**8
enddo
```

```
sum=sum/float(k)
expression=sum
end
subroutine prog1(n,x,y,z)
integer term1,term2
dimension x(n),y(n+1),z(n)
dimension xloc(10)
common/c1/term1,term2
do i=1,n
  z(i)=x(i)+y(i)
endo
do i=1,10
  z(i)=z(i)*ztab(i)
enddo
facteur=float(term1*term2)
do i=1,n
  z(i)=z(i)*facteur
enddo
end
```

Sous-programme prog1: noter comment les dimensions des tableaux passés par argument sont déclarés. Le tableau xloc(10) est un tableau local. Ses dimensions ne peuvent pas être définies par une variable passée par argument (variable non locale).

C. Quelques éléments d'Unix.

Tout comme le Fortran, le système d'exploitation (SE) Unix est devenu une quasi-norme dans la communauté scientifique. Unix est un SE très simple à utiliser. En fait, l'idée fondamentale d'Unix est que tout objet traité est considéré soit comme un fichier (un fichier Fortran, une commande exécutable du système, l'écran, une imprimante, etc...), soit comme un processus (commande en cours d'éxécution).



tous vos fichiers

Le “Home Directory” est le directory (ensemble de fichiers et de directories) dans lequel vous vous trouvez quand vous vous “loggez” (connectez à la machine). Pour savoir où vous êtes:

```
$ pwd
/usr/vous
```

Le signe “$” est ce qu’on appelle le prompteur. Le signe représentant le prompteur peut changer d’une machine à l’autre et il peut être redéfini par l’utilisateur. Dans votre drectory vous pouvez faire ce que vous voulez mais évidemment pas ailleurs. Pour monter d’un niveau dans l’arbre des fichiers:

```
$cd ..
$pwd
/usr
```

Pour descendre d’un niveau:

```
$cd vous
$pwd
/usr/vous
```

vous êtes revenus chez vous. Vous pouvez aller partout dans le système. Pour connaître la

liste des fichiers ou des directories à l'endroit où vous vous trouvez, faire la commande:
$ls
En fait, pour plus de détails sur les fichiers:
$ls -l
Pour connaître le mode d'emploi de toute commande Unix vous pouvez utiliser le manuel en ligne en tapant "man" suivi du nom de la commande. "ls -l" permet de savoir, en autres choses, si on a affaire à un fichier ou à un directory.
Copie d'un fichier:
$cp fic1 fic2
Changement de nom:
$mv fic1 fic2
Destruction d'un fichier:
$rm fic1
etc... (voir livre quelconque sur Unix).

2. Processus

Pour lancer un processus on lance une commande qui n'est rien d'autre qu'un fichier exécutable. Le fichier exécutable peut être soit un fichier exécutable système (comme: ls,cp,mv,etc...) ou un fichier exécutable que vous avez fabriqué. L'exemple le plus courant pour nous sera la création d'un fichier exécutable issu du compilateur Fortran:
$xlf fichier.f
xlf est le nom du compilateur Fortran sur la machine que vous utiliserez (IBM Risc). Ce nom peut évidemment varier d'une machine à l'autre. Noter que tout fichier soumis au compilateur Fortran doit avoir un nom se terminant par ".f" xlf crée un fichier qu'il appelle a.out par défaut. "a.out" est une commande exécutable:
$a.out
lance le programme. La commande "ps" permet d'avoir quelques formations sur les processus en cours d'exécution. La commande "kill" permet de tuer un processus. La commande "kill" doit être suivie du numéro du processus à tuer donné par la commande "ps". Noter aussi l'existence de la commande "time" qui permet de savoir le temps total d'exécution de la commande concernée. Par exemple:
$time a.out
vous permet de connaître le temps d'exécution de votre programme.

D. L'éditeur de texte "vi"

Pour écrire un texte dans un fichier il faut un éditeur de texte. Il existe de nombreux éditeurs de texte plus ou moins sophistiqués. Sous Unix, il existe un éditeur standard que vous retrouverez sous tous les systèmes, partout dans le monde. Il s'agit de l'éditeur de texte 'vi' qu'il faut donc, à mon avis, absolument connaître. Avec ce document est joint un document complet sur vi. Je ne mentionnerai donc que le point important suivant. Il existe essentiellement deux modes sous vi:
1. Le mode normal et initial qui vous permet de visualiser le texte, de positionner le curseur

où vous voulez et d'effectuer un certain nombre de commandes.
2. le mode dit d'insertion dans lequel on entre en tapant la lettre "i". Vous pouvez alors entrer du texte à partir de l'endroit où se trouve le curseur. Pour sortir du mode insertion il faut appuyer sur la touche "Echappement" (Escape)

E. Quelques conseils de programmation

L'expérience montre qu'il est très important d'avoir une discipline stricte de programmation. Ceci permet de gagner un temps énorme et d'éviter de longues heures et jours de recherche de "bugs" qui peuvent facilement être évités. Evidemment, la bonne stratégie à employer est une question très personnelle et, en général, chacun est persuadé d'avoir la meilleure méthode (n'argumentez pas, ça ne sert à rien!). Essayer de développer très vite votre propre stratégie et copier que ce qui vous convient chez les autres. Quelques conseils (faites en ce que vous voulez...):

Il faut programmer vite

L'objectif n'est pas de programmer à la perfection et de faire la simulation ultime, mais d'étudier un problème physique et d'utiliser l'ordinateur comme un outil et non pas comme une fin en soi. Plus vous programmerez vite, plus vous serez disponible pour le problème à étudier, et plus vous serez psychologiquement prêts à utiliser sans hésiter l'ordinateur pour attaquer un problème nouveau. Si vous devez prendre des semaines pour mettre en place votre algorithme, vous abandonnerez très vite.

Il faut programmer juste

Programmer juste, c'est à mon avis développer une stratégie sophistiquée de VERIFICATION de votre programme. Pour cela, les points suivants me paraîssent importants:

i. Avant d'écrire quoi que ce soit, murissez l'algorithme dans votre tête aussi longtemps que la structure générale n'est pas claire. Ne vous jettez pas sur la machine.

ii. Décomposer votre programme en autant d'éléments élémentaires que possible. Tester chacune des sous-parties séparément. Tester les sous-programmes *in situ*, c'est à dire dans votre programme principal. Il arrive souvent que le sous-programme marche quand il est traité séparement mais donne des résultats faux dans le programme principal (par exemple, variables globales qui interfèrent).

iii. Ecrivez un seul programme, éviter les multiples versions (monprog.f, monprog1.f, monprog2.f, monprog2a.f, etc...). Eviter de traiter les cas particuliers séparément, passer au cas le général tout de suite. Ne multiplier pas les commons, les include, les fichiers d'entrée et de sortie.

iv. Vérifiez tous les cas-limites connus exactement que votre programme général peut produire. Essayer de comprendre chaque décimale (une erreur relative de 10^-6 sur un résultat en double précision est toujours instructive). N'oubliez pas que le chaos a été découvert de cette façon!

Une rêgle très importante est que le temps que vous passez à vérifier votre programme doit être beaucoup plus long que le temps que vous avez passé à l'écrire.

Il faut programmer de manière systématique

Choisissez la méthode que vous voulez, mais soyez systématique. Par exemple:

Programme Fortran:

```
      program general
      include 'general.data'
      dimension x(nmax), etc....
      .....
      print*,'n?'
      read*,n
      if(n.gt.nmax)stop 'nmax dans general.data trop petit'
      .....
      call prog1(....)
      .....
      end
CC Ici, vous donnez des explications détaillées de ce qui est fait
CC dans le sous-programme prog1.  Il faut absolument donner
CC TOUS les détails (formule exacte, conventions multiples, références
CC dans la littérature, etc...)
CC sinon ça ne sert absolument à rien:
CC
CC input:
CC ******
CC    liste des quantités utilisées comme input et non modifiées
CC
CC output:
CC ******
CC    liste des quantités calculées et modifiées
CC
      subroutine prog1(....)
      include 'general.data'
      .....
      end
```

Fichier "general.data" inséré par include dans TOUTES les subroutines et fonctions:

```
      implicit double precision(a-h,o-z)
** Parameter pour dimensions de TOUS les tableaux du programme:
      parameter(nmax=100,....)
      common/c1/x1(nmax),....
      ....
```

Commentaires:

- L'ordre include permet d'introduire le fichier ayant le nom "general.data" à l'endroit où l'ordre include se trouve. Il faut absolument introduire cet include dans TOUS les sous-programmes et fonctions du programme. Ceci vous évitera un nombre incalculables d'erreurs.

- Utilisation de l'ordre: "implicit double precision(a-h,o-z)". Il peut être intéressant d'utiliser un ordre "implicit none" au départ et de préciser ensuite le type de chacune des quantités introduites dans le programme. Si vous programmez, vous vous rendrez compte très vite que, si vous adoptez la convention précédente (un seul fichier include en début de TOUS les sous-programmes et TOUTES les fonctions), vous pouvez utiliser la convention par défaut du Fortran sans aucun danger et ne vous ferez JAMAIS d'erreur de type qui sont, l'expérience le montre, les erreurs les plus courantes.

- Parameter(nmax=100,...): les dimensions de TOUS les tableaux de votre programme doivent être paramétrisées et la dimension doit absolument se trouver dans l'include unique. J'insiste: ne jamais écrire quelque chose comme: "X(100) " dans un programme, même s'il est évident que vous ne changerez jamais la dimension du tableau X.

- programme principal: if(n.gt.nmax)stop 'nmax dans general.data trop petit'. Ceci est absolument impératif. Si une donnée d'entrée conditionne la taille d'un tableau, faire un test de ce type.

- Il faut absolument adopter une politique cohérente pour le passage d'une quantité d'un sous-programme à l'autre. Il y a deux façons de faire cela. Par common ou par argument. Je propose de faire passer par common (common qui doit se trouver dans le fichier include et pas ailleurs) toute quantité qui est commune à plus de deux sous-programmes. Les autres quantités doivent passer par argument entre le programme appelant et le programme appelé.

F. Aspects finis de l'ordinateur

Il est très important de bien comprendre les aspects finis de la machine. Nous allons les illustrer sur des exemples simples.

1. Vitesse finie

On se propose d'estimer l'ordre de grandeur de la vitesse du processeur. Pour cela, on va faire tourner un petit programme calculant le nombre π à l'aide de la série infinie suivante

(ce n'est pas la série la plus rapidement convergente pour calculer π!):

$$\frac{\pi^2}{6} = \lim_{n\to+\infty} \sum_{k=1}^{n} \frac{1}{k^2}$$

Ecrivez un programme qui permet de calculer π, calculer le nombre d'opérations élémentaires flottantes effectuées dans votre programme pour un n maximum donné. Faites tourner le programme et mesurez les temps d'exécution à l'aide de la commande 'time'. Une solution possible:

```
program calcpi
implicit double precision (a-h,o-z)
print*,'n?'
read*,n
nsort=n/10
pi=0.d0
do i=1,n
  pi=pi+1.d0/dfloat(i)**2
  if(mod(i,nsort).eq.0)print*,i,dsqrt(pi*6.d0)
enddo
end
```

Effectuer les calculs avec ou sans l'ordre de sortie. Ordres de grandeur?

2. Précision finie

Les calculs avec des entiers sont exacts sur ordinateur. Ceci n'est pas le cas avec des réels. On se propose de calculer la solution générale de l'équation aux différences finies suivante:

$$x_n = 37x_{n-1}/6 - x_{n-2}$$

avec :

$$x_0 = 1 \text{ et } ; x_1 = 1/6$$

On peut montrer facilement que la solution est:

$$x_n = 1/6^n$$

Ecrire un programme qui calcule:

1. Itérativement la solution de l'équation précédente
2. Directement la solution connue.
3. Analyser l'évolution de l'erreur en fonction de n. Faites cette étude en simple précision, en double précision, et en quadruple précision (la quadruple précision peut être obtenue sur votre machine IBM en compilant le programme écrit en double precision à l'aide de l'option de compilation suivante: xlf -qautodbl=dblpad programme.f).

Une solution possible:

```
program simpleprecision
```

```
print*,'n?'
read*,n
if(n.le.1)stop 'n plus grand que 1 SVP'
x0=1.
x1=1./6.
do i=1,n-1
  x2=37.*x1/6.-x0
  x0=x1
  x1=x2
enddo
print*,'n= ',n
print*,'xn calcule iterativement= ',x2
print*,'xn solution exacte= ',1./6.**n
end
program doubleprecision
implicit double precision(a-h,o-z)
print*,'n?'
read*,n
if(n.le.1)stop 'n plus grand que 1 SVP'
x0=1.d0
x1=1.d0/6.d0
do i=1,n-1
  x2=37.d0*x1/6.d0-x0
  x0=x1
  x1=x2
enddo
print*,'n= ',n
print*,'xn calcule iterativement= ',x2
print*,'xn solution exacte= ',1.d0/6.d0**n
end
```

3. Temps d'accès finis à la mémoire centrale et à la mémoire dure

En faisant tourner les deux-programmes suivants et en utilisant la commande 'time', évaluer grossièrement le rapport entre le temps d'accès moyen à un élément de mémoire centrale et à un élément de mémoire dure.

Premier programme:

```
program temps1
print*,'n?'
read*,n
open(1,file='ecriture')
rewind1
do i=1,n
```

```
  do j=1,10000
    write(1,*)j
  enddo
  rewind1
enddo
end
```

Deuxième programme:

```
program temps2
parameter(nmax=10000)
dimension x(nmax)
print*,'n?'
read*,n
do i=1,n
  do j=1,10000
    x(j)=j
  enddo
enddo
end
```

III. DYNAMIQUE MOLÉCULAIRE. CHAOS CLASSIQUE

A. Généralités

1. Equations différentielles

Une équation différentielle générale peut toujours se réduire à un système de N équations différentielles du 1^{er} ordre couplées.

$$\frac{dy_i(t)}{dt} = f_i'(t, y_1, ..., y_N) \quad i = 1, .., N$$

plus des conditions aux limites qui jouent un rôle important. On peut distinguer deux classes de conditions aux limites:

i. Conditions aux valeurs initiales:

Exemple: les valeurs de $y_1(0), ..., y_N(0)$ sont imposées

ii. Conditions aux limites:

Exemple: les valeurs de $y_1(0), ..., y_k(0), y_{k+1}(t), ..., y_N(t)$ sont imposés

Il existe de nombreuses méthodes pour intégrer le système précédent: Runge-Kutta, extrapolation de Richardson, etc... (voir Numerical Recipes). Cependant, toutes les méthodes sont construites à partir de la même idée élémentaire. Illustrons cette idée avec le système le plus simple qu'on puisse imaginer. On veut résoudre l'équation:

$$y'(t) = f(y, t)$$

On utilise une représentation discrétisée de la dérivée, ainsi que l'équation donnant l'expression de la dérivée:

$$\frac{y(t+\tau)-y(t)}{\tau} = f(y(t),t) + O(\tau)$$

et on peut donc écrire:

$$y(t+\tau) = y(t) + \tau f(y(t),t) + O(\tau^2)$$

Si y(0) est connue alors la trajectoire-solution peut être construite de manière *itérative* et sans *stockage d'information.*

On peut augmenter la précision en diminuant τ et/ou en faisant intervenir autant de dérivées d'ordre supérieurs que l'on désire. Par exemple, si on veut pousser la précision à un ordre supérieur:

$$y(t+\tau) = y(t) + \tau y' + \frac{\tau^2}{2} y'' + O(\tau^3)$$

c'est à dire:

$$y(t+\tau) = y(t) + \tau f(y(t),t) + \frac{\tau^2}{2} f'(y(t),t) + O(\tau^3)$$

et ainsi de suite...

2. Dynamique moléculaire

On souhaite intégrer les équations de Newton, c'est à dire 2DN équations différentielles couplées où D est la dimension de l'espace et N, le nombre de particules:

$$\begin{cases} \vec{v}_i(t) = \frac{d\vec{r}_i}{dt} \\ \\ \vec{a}_i(t) = \frac{\vec{F}_i}{m_i} \end{cases} \quad \text{i=1,N}$$

avec l'ensemble de conditions initiales: $\{\vec{v}_i(0), \vec{r}_i(0)\}_{i=1,N}$

Algorithme du 1^{er}ordre:

$$\begin{cases} \vec{r}_i(t+\tau) = \vec{r}_i(t) + \tau\vec{v}_i(t) + O(\tau^2) \\ \\ \vec{v}_i(t+\tau) = \vec{v}_i(t) + \tau\vec{a}_i(t) + O(\tau^2) \end{cases} \quad \text{i=1,N}$$

ou sa généralisation à un ordre quelconque.

Algorithme de Verlet (ou "leapfrog", "saut de grenouille")

$$\begin{cases} \vec{r}_i(t+\tau) = \vec{r}_i(t) + \tau\vec{v}_i(t) + \frac{1}{2}\tau^2\vec{a}_i(t) \\ \\ \vec{v}_i(t+\tau) = \vec{v}_i(t) + \frac{\tau}{2}(\vec{a}_i(t) + \vec{a}_i(t+\tau)) \end{cases} \quad \text{i=1,N}$$

Vérifier qu'il s'agit d'un algorithme du 2^{ime} ordre.

a.**Nombre de degrés de liberté égal à un (D=1)**

Un système Hamiltonien à un degré de liberté est intégrable, c'est à dire les trajectoires peuvent être obtenues par de simples quadratures. On a:

$$\begin{cases} \frac{dx}{dt} = v(t) \\ \frac{dv}{dt}(t) = F(x(t))/m \\ F(x) = -\frac{\partial V}{\partial x} \end{cases}$$

L'énergie est évidemment une intégrale première du mouvement:

$$\frac{dE}{dt} = 0$$

d'où

$$\frac{dx}{dt} = \pm\sqrt{\frac{2}{m}(E - V(x))}$$

et on a donc:

$$t - t_0 = \pm \int_{x_0}^{x} \frac{dx}{\sqrt{\frac{2}{m}(E - V(x))}},$$

ce qui permet d'évaluer la trajectoire x(t).

b.**Nombre de degrés de liberté plus grand que un (D> 1)**

L'intégrabilité correspond à l'existence de (D-1) intégrales du mouvement en plus de l'énergie totale.

Méthodes des sections de Poincaré:

Cette méthode permet de déceler l'existence d'intégrales premières du mouvement supplémentaires. Elle consiste à visualiser des coupes de l'hypersurface à énergie constante. Nous allons illustrer cette idée dans un cas à deux degrés de liberté. Considérons, par exemple, la section de Poincaré correspondant à $v_x = 0$ avec $v_y > 0$ (choix du sens du temps). S'il existe une intégrale première, I, en plus de l'énergie, on a:

$$\mathrm{I(x, y, v_x, v_y)} = \mathrm{cte}$$

C'est à dire que l'on peut écrire: $E(x, y, 0, v_y)$=cte et $I(x, y, 0, v_y)$=cte, ce qui implique qu'il existe une relation implicite f(x,y)=0. Donc, dans le cas où le système à une intégrale supplémentaire, les sections de Poincaré doivent être des courbes continues (éventuellement non connexes) dans le plan (x,y). L'existence de sections de Poincaré discontinues à l'allure chaotique s'oppose donc à l'existence d'une intégrale première supplémentaire.

Exemple de programme Fortran:

```
      program dynmol
CC Input:
      print*,'Tau?  Nsteps?'
      read*,tau,nsteps
      print*,'x0 y0 vx0 vy0'
      read*,xold,yold,vxold,vyold
CC Calcul de l'énergie initiale:
      e0=0.5*(vxold**2+vyold**2)+v(xold,yold)
      print*,'Energie=',e0
CC Calcul de l'accélération initiale:
      call computea(xold,yold,axold,ayold)
CC Construction itérative de la trajectoire:
      do k=1,nsteps
        xnew=xold+tau*vxold+0.5*tau**2*axold
        ynew=yold+tau*vyold+0.5*tau**2*ayold
        call computea(xnew,ynew,axnew,aynew)
        vxnew=vxold+0.5*tau*(axold+axnew)
        vynew=vyold+0.5*tau*(ayold+aynew)
        e0=0.5*(vxnew**2+vynew**2)+v(xnew,ynew)
        print*,'t  x,y  energie'
        print*,float(k)*tau,xnew,ynew,e0
        xold=xnew
        yold=ynew
        vxold=vxnew
        vyold=vynew
        axold=axnew
        ayold=aynew
      enddo
      print*,'Simulation terminee'
      end
      subroutine computea(x,y,ax,ay)
      ax=-x-2.*y**2*x
      ay=-y-2.*x**2*y
      end
      function v(x,y)
      v=0.5*(x**2+y**2)+x**2*y**2
      end
```

B. Apparition du chaos dans un système d'oscillateurs couplés

I. On se propose d'observer le développement du chaos pour un système de deux oscillateurs couplés décrit par le potentiel:

$$V(x,y) = \frac{1}{2}x^2 + \frac{1}{2}y^2 + x^2y^2 \tag{1}$$

l'énergie étant donnée par:

$$E = \frac{1}{2}(v_x^2 + v_y^2) + V(x,y) \quad (\text{masses} = 1) \tag{2}$$

1. Ecrivez un programme qui met en oeuvre l'algorithme de Verlet pour intégrer les équations classiques du mouvement (on travaillera en double précision). On rappelle que l'algorithme s'écrit:

$$\vec{r}(t+\tau) = \vec{r}(t) + \tau\vec{v}(t) + \frac{1}{2}\tau^2\vec{a}(t)$$

$$\vec{v}(t+\tau) = \vec{v}(t) + \frac{\tau}{2}(\vec{a}(t) + \vec{a}(t+\tau)) \tag{3}$$

où $\vec{r} = (x,y), \vec{v} = (v_x, v_y)$, $\vec{a} = (-\frac{\partial V}{\partial x}, -\frac{\partial V}{\partial y})$ et τ le pas temporel choisi.

2. Construire une trajectoire en prenant des conditions initiales $x^{(0)}, y^{(0)}, v_x^{(0)}$ aléatoires, $v_y^{(0)}$ étant choisi de manière à imposer la valeur de l'énergie E désirée: $v_y = \sqrt{2(E - V(x^{(0)}, y^{(0)}) - \frac{1}{2}v_x^2)}$.

3. Analyser le comportement numérique de l'énergie en fonction du nombre de pas et du pas temporel. Ecrire un système d'équations pour les équations du mouvement admettant une erreur d'intégration un ordre de grandeur plus petit en τ. Retrouver ce résultat numériquement (on illustrera ces resultats à l'aide de courbes). Quelle est la valeur de τ optimale pour un calcul en double precision ?

4. Tracer une trajectoire (x(t),y(t)) à l'aide du programme graphique mis à votre disposition.

5. On se propose maintenant de réaliser une "section de Poincaré", c'est à dire une coupe particulière de l'hypersurface (ici, à 3 dimensions) à énergie constante. On considèrera la coupe définie par: $v_x = 0$ avec $v_y > 0$ (choix du sens du temps le long de la trajectoire). En pratique, on écrira dans un fichier les points (x,y) de la trajectoire vérifiant: $|v_x| < \epsilon$ et $v_y > 0$. Afin de disposer d'un nombre de points suffisant, on prendra les paramètres suivants: $\tau = 0.0001, \epsilon = 0.0001$ et quelques millions de pas d'intégration.

6. Visualiser les sections de Poincaré pour différentes valeurs de l'énergie (E variant de 1 à 2). Observer la modification brutale de la section en fonction de l' énergie correspondant au passage d'un régime quasi-intégrable à un régime de chaos fort. Expliquer comment on peut comprendre ce phénomène en raisonnant sur le nombre d'intégrales premières du mouvement.

7. Refaire la même étude pour un système de deux oscillateurs couplés décrit par le potentiel:

$$V(x,y) = \frac{1}{2}x^2 + \frac{1}{2}y^2 + \frac{1}{4}(x^4 + y^4) - x^3y - xy^3 + \frac{3}{2}x^2y^2 \tag{4}$$

Discuter les résultats que vous obtenez.

II. On se propose maintenant d'étudier un système non conservatif représentant un oscillateur entretenu et décrit par l'équation dite de van der Pol:

$$\frac{d^2\theta}{dt^2} - (\epsilon - \theta^2)\frac{d\theta}{dt} + \theta = 0 \tag{5}$$

1. Ecrire un programme qui intégre les équations du mouvement.
2. Montrer qu'il existe un attracteur (cycle-limite) pour les trajectoires dans l'espace des phases $(\dot{\theta}, \theta)$.

IV. MÉTHODES DE MONTE CARLO. TRANSITION DE PHASE DANS LE MODÈLE D'ISING À 2 D

A. Méthodes de Monte Carlo

Le coeur des méthodes de Monte Carlo repose sur l'évaluation numérique d'intégrales multidimensionnelles de la forme:

$$\mathrm{I} = \int_{\mathcal{E}} dx \Pi(x) f(x)$$

x $\in \mathcal{E}$, espace de configuration ou espace des états. Le signe intégrale représente ici aussi bien une intégrale qu'une somme discrète. Donnons quelques exemples: $\mathcal{E}$=R^d, R^{3N} où N est un nombre de particules, $\mathcal{E}$=2^{N_s} où N_s est un nombre de sites pour un système à deux états par site, etc ... Π(x) est une densité de probabilité, c'est à dire:

$$\Pi(x) \geq 0 \quad \text{et} \quad \int_{\mathcal{E}} dx\ \Pi(x) = 1$$

L'algorithme le plus utilisé pour calculer de telles intégrales est l'algorithme de Metropolis. Cet algorithme est un algorithme probabiliste qui permet d'engendrer des configurations $x^{(i)}$ selon la densité $\Pi(x)$. Ainsi, l'intégrale I peut être évaluée sous la forme:

$$\mathrm{I} = \lim_{P\to\infty} \frac{1}{P} \sum_{i=1}^{P} f(x^{(i)})$$

En pratique,

$$\mathrm{I} = \frac{1}{\mathrm{P}} \sum_{\mathrm{i}=1}^{\mathrm{P}} \mathrm{f}(\mathrm{x}^{(\mathrm{i})}) + \frac{\mathrm{c}}{\sqrt{\mathrm{P}}} \quad \text{pour P assez grand}$$

où $\{x^{(i)}\}_{i=1,P}$ est un ensemble de P configurations construites avec l'algorithme de Metropolis, et où $\frac{c}{\sqrt{P}}$ représente l'erreur résiduelle (théorème de la limite centrale, loi des grands nombres). On va voir qu'une caractéristique essentielle de l'algorithme est qu'il ne requiert que la connaissance de probabilités relatives, c'est à dire, $\frac{\Pi(x^{(i)})}{\Pi(x^{(j)})}$. Ceci est particulièrement intéressant car les intégrales à évaluer mettent souvent en jeu une probabilité de la forme:

$$\Pi(x) = \frac{\exp(-\Phi(x))}{\int dx \exp(-\Phi(x))}$$

où $\Phi(x)$ est connue explicitement, la norme $\int dx \exp(-\Phi(x))$ étant inconnue (intégrale multidimensionnelle compliquée). Exemple: ensemble canonique classique où $\Phi = \beta H$, H étant la fonction énergie totale du système.

B. Algorithme de Metropolis

Sans perdre de généralité nous nous placerons dans le cas d'un espace de configuration ayant un nombre fini, N, de configurations possibles.

- Déf.1 Probabilité:
  $\Pi_i \geq 0$ i=1,N
- Déf.2 Probabilité de transition (ou matrice stochastique) $P_{i\to j}$:

  i. $P_{i\to j} \geq 0$

  ii. $\sum_{j=1}^{N} P_{i\to j} = 1$ (indépendant de i)
- Déf.3 Probabilité de transition ergodique:
  $\forall i_0 \ \ \forall i$ il existe une probabilité non nulle qu'après un nombre *fini* d'application de la probabilité de transition, partant de l'état i_0 on arrive à l'état i.

Enonçons maintenant l'algorithme de Metropolis:

Soit $P^*_{i\to j}$ une probabilité de transition ergodique, alors $P_{i\to j}$ définie de la manière suivante:

$$\begin{cases} P_{i\to j} = P^*_{i\to j}\mathrm{Min}(1, R_{ij}) & \mathrm{j} \neq \mathrm{i} \\ P_{i\to i} = P^*_{i\to i} + \sum_{k\neq i} P^*_{i\to k}(1 - \mathrm{Min}(1, R_{ik})) & \mathrm{j} = \mathrm{i} \\ \text{avec} \ \ R_{ij} = \frac{\Pi_j P^*_{j\to i}}{\Pi_i P^*_{i\to j}} \end{cases}$$

est une probabilité de transition ergodique admettant Π_i comme probabilité stationnaire.

Démonstration:

1. Probabilité de transition:
   - $P_{i\to j} \geq 0$: évident
   - $\sum_{j=1}^{N} P_{i\to j} = \sum_{j\neq i} P_{i\to j} + P_{i\to i}$
     $= \sum_{j\neq i} P^*_{i\to j} + P^*_{i\to i}$
     $= 1$

2. Probabilité stationnaire:
   Il faut montrer que: $\sum_i \Pi_i P_{i\to j} = \Pi_j$ Pour cela, nous allons montrer tout d'abord que le couple: $\{P_{i\to j}; \Pi_i\}$ vérifie la condition dite de bilan détaillé:

$$\Pi_i P_{i\to j} = \Pi_j P_{j\to i} \quad \forall (i,j)$$

   Démonstration:

   - i=j égalité évidente
   - i≠j: le rapport des deux membres de l'égalité précédente vaut:

$$\frac{\Pi_j P_{j\to i}}{\Pi_i P_{i\to j}} = \frac{R_{ij}\mathrm{Min}(1, R_{ji})}{\mathrm{Min}(1, R_{ij})}$$

     en notant que $R_{ij} = 1/R_{ji}$ et en distinguant les deux cas correspondant à $R_{ij} \geq 1$ et $R_{ij} < 1$, on vérifie facilement que ce rapport vaut un.

   Finalement, en utilisant la propriété de bilan détaillé, on a:

$$\sum_i \Pi_i P_{i\to j} = \sum_i \Pi_j P_{j\to i} = \Pi_j$$

   ce qui montre que Π_i est probabilité stationnaire.

   La propriété d'ergodicité est très importante car elle exprime le fait que tout état de l'espace de configuration peut être atteint en un nombre de pas fini (tous les états sont "visités"), et, en fait, cette condition est nécessaire dans la démonstration précédente afin éviter le cas pathologique que nous n'avons pas considéré où la probabilité de transition vers un état donné j s'annule quelque soit l'état i.

On peut démontrer le théorème suivant qui précise la notion de convergence. Soit $f^{(k)}$ une distribution, c'est à dire un ensemble de N nombres positifs. On va écrire l'application de la probabilité de transition à cette distribution sous la forme:

$$f_i^{(k+1)} = \sum_j f_j^{(k)} P_{j\to i} \equiv P f_i^{(k)}$$

On a le résultat suivant:

$$\lim_{n\to\infty} f_i^{(n)} \sim P^n f_i^{(0)} = \Pi_i \quad \forall f^{(0)}$$

Les différentes étapes de la démonstrations sont les suivantes:

- On associe à $P_{i\to j}$ une matrice réelle *symétrique* définie de la manière suivante:

$$M_{ij} = \sqrt{\Pi_i} P_{i\to j} \frac{1}{\sqrt{\Pi_j}}$$

  Notons que la probabilité de transition elle-même ne définit pas une matrice symétrique.

- On vérifie facilement que $\sqrt{\Pi}$ est état propre de M avec valeur propre 1:

$$\sum_j M_{ij}\sqrt{\Pi_j} = \sqrt{\Pi_i}$$

- On vérifie aussi que:

$$P^n f^{(0)} = \sqrt{\Pi} M^n \frac{f^{(0)}}{\sqrt{\Pi}}$$

- On utilise la décompostion spectrale de M qui nous dit que, dans la limite des grandes valeurs de n, la matrice se réduit au projecteur sur le sous-espace des vecteurs propres associé à la plus grande valeur propre. On peut montrer que la matrice M, du fait de sa structure, a des valeurs propres inférieures ou égales (en module) à 1 et que dans le cas où la densité Π ne s'annule pas le sous-espace associé n'est pas dégénéré. En conséquence:

$$P^n f^{(0)} = c\,\Pi$$

où c est un coefficient qui représente le recouvrement de la distribution $f^{(0)}/\sqrt{\Pi}$ et du vecteur propre $\sqrt{\Pi}$ de la matrice M.

C. Implémentation pratique

On se donne $P^*_{i\to j}$ une probabilité de transition appelée dans la suite "probabilité de transition d'essai". On suppose que l'on sait échantillonner cette probabilité, c'est à dire étant donnée une configuration i, tirer des configurations aléatoires j distribuées selon la loi: $P^*_{i\to j}$. Comment passer d'une configuration à la suivante selon la probabilité de transition $P_{i\to j}$ définie précedemment? Les étapes sont les suivantes:

- Soit i la configuration courante. On effectue un "mouvement d'essai" ("trial move") dans l'espace des états en tirant une nouvelle configuration j selon $P^*_{i\to j}$.

- On calcule le rapport $R_{ij} = \frac{\Pi_j P^*_{j\to i}}{\Pi_i P^*_{i\to j}}$. Le mouvement d'essai est accepté (c'est à dire la nouvelle configuration est j) avec probabilité $q = \mathrm{Min}(1, R_{ij})$. Si le mouvement est refusé, alors la nouvelle configuration est l'ancienne, c'est à dire la configuration i.

Accepter avec probabilité q est réalisé en comparant q à un nombre aléatoire uniforme ξ tiré entre 0 et 1. Si $q > \xi$ le mouvement est accepté, sinon le mouvement est refusé.

Attention!! Il est important de souligner que, lorsque le mouvement est refusé, l'ancienne configuration doit bien être considérée comme une nouvelle configuration; en particulier, sa contribution aux intégrands doit être ajoutée comme toute autre nouvelle configuration.

Algorithme de Metropolis classique:

J'appelle ici "algorithme de Metropolis classique" l'algorithme qui est généralement présenté dans les ouvrages et qui correspond en fait à un choix particulier de probabilité de transition d'essai:

$$\mathrm{P}^*_{\mathrm{i}\to\mathrm{j}} = \text{constante}$$

pour un sous-ensemble d'états j "voisins" dans l'espace de configurations (notion de voisinage dépend du problème). Pour cet algorithme on a donc: $R_{ij} = \frac{\Pi_j}{\Pi_i}$.

Cet algorithme est généralement exprimé pour un espace de configuration continu. Notons $x = (x_1, x_2, ..., x_d)$ un élément de l'espace des états, la probabilité de transition s'écrit formellement:

$$P^*_{x \to y} = \frac{\Theta_\Delta(y - x)}{\Delta^d}$$

où Θ_Δ est la fonction caractéristique de l'hypercube centré en 0 et de coté de longueur Δ. En pratique, cela veut dire que les mouvements d'essai sont effectués de la manière suivante:

$$y_i = x_i + \Delta(\xi_i - 0.5) \quad i = 1, d$$

où ξ_i sont des nombres aléatoires uniformes tirés entre 0 et 1. Δ est une quantité positive *a priori* arbitraire comme le choix de toute probabilité de transition d'essai. Cependant, Δ détermine la vitesse de convergence vers la densité stationnaire. Prenons les deux cas-limites possibles pour Δ. Quand Δ est choisi très petit, le rapport R_{ij} est toujours très voisin de l'unité et les mouvements d'essai sont acceptés avec une grande probabilité. Malheureusement, l'espace de configuration est visité avec inefficacité et l'estimateur des intégrales convergera lentement. Dans la limite opposée, il est clair que, dans un espace multidimensionnel, un mouvement de grande amplitude aboutira quasiment toujours à un refus. Il y a donc un optimum à trouver. Cet optimum correspond à un choix de Δ donnant un taux d'acceptation fini, ni trop petit, ni trop grand, un taux voisin de 0.5 est souvent conseillé.

Appliquons cet algorithme à un cas simple. Supposons que l'on désire évaluer l'intégrale suivante:

$$I(g) = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{+\infty} dx \exp(-x^2 - gx^4)$$

Cette intégrale n'a pas de valeur analytique connue pour g différent de 0. Nous allons utiliser l'algorithme de Metropolis pour évaluer cette intégrale. Evidemment, c'est la pire des méthodes à utiliser pour ce type d'intégrale, mais ceci nous donnera l'occasion d'exemplifier ce que nous venons dire dans un cas très simple. On prendra comme densité $\Pi(x) = \frac{1}{\sqrt{\pi}} \exp(-x^2)$.

Proposition de programme Fortran:

```
      program Metropolis
CC Input:
      print*,'delta?  nsteps?g?'
       read*,delta,nsteps,g
CC Configuration initiale tirée au sort:
       xold=ranf()-0.5
CC Initialisation des accumulateurs:
       valint=0.
       naccep=0
CC Génération des configurations successives:
```

```
do k=1,nsteps
  xnew=xold+(ranf()-0.5)*delta
  ratio=proba(xnew)/proba(xold)
  if(ratio.ge.ranf())then
    naccep=naccep+1
  else
    xnew=xold
  endif
  valint=valint+f(xnew,g)
  xold=xnew
enddo
valint=valint/float(nsteps)
print*,'Integrale= ',valint
tauxaccep=float(naccep)/float(nsteps)
print*,'Taux acceptation= ',tauxaccep
print*,'Simulation terminee'
end
function proba(x)
proba=exp(-x**2)
end
function f(x,g)
f=exp(-g*x**4)
end
```

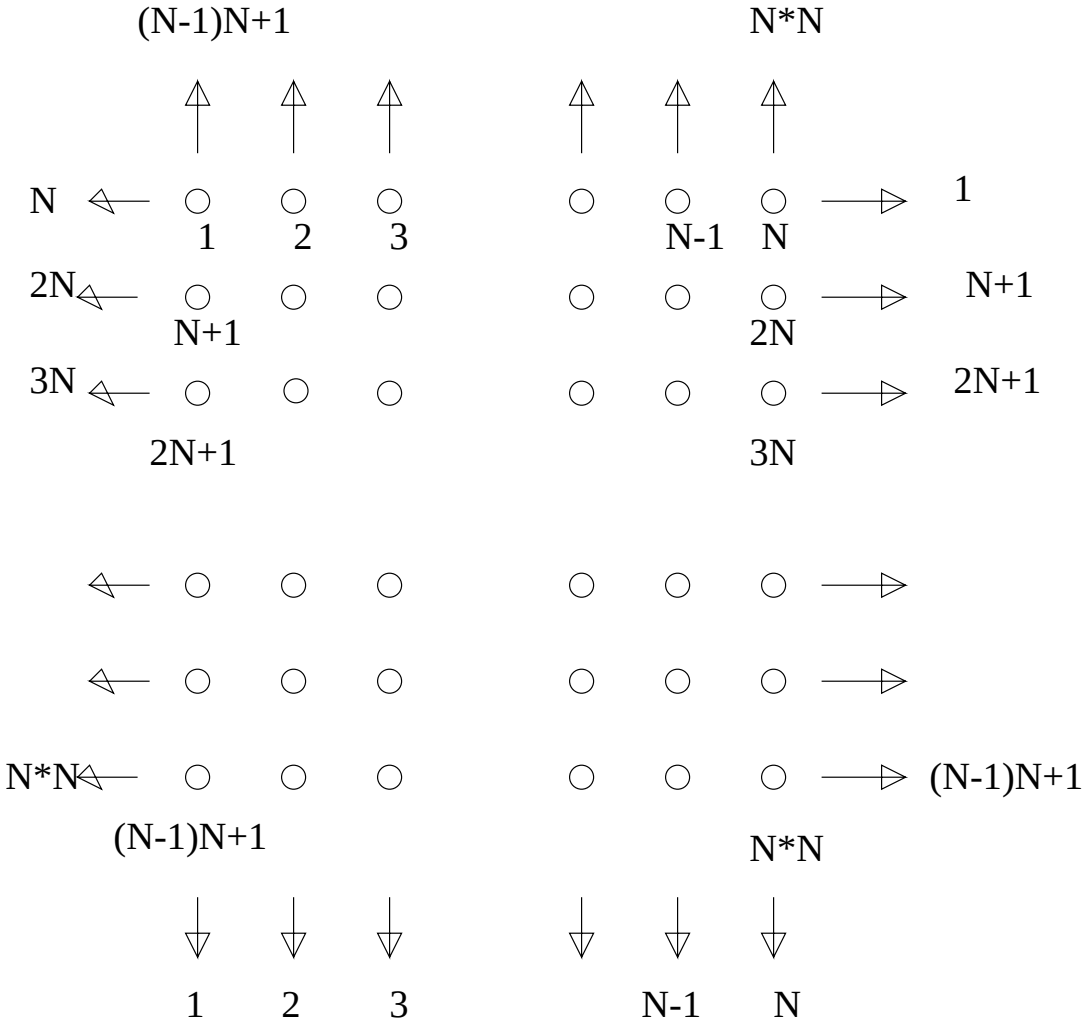
Ici, ranf() désigne une fonction intrinsèque délivrant un nombre aléatoire compris entre 0 et 1 avec une distribution uniforme. En général, les machines disposent toujours d'une fonction de ce type dans leur bibliothèque mathématique de base (évidemment, le nom de la fonction peut être différent d'une machine à l'autre).

D. Simulation du modèle d'Ising à 2 dimensions

On se propose d'écrire un programme pour calculer l'énergie moyenne et la chaleur spécifique du modèle d'Ising à deux dimensions.
Le modèle est décrit par la fonction énergie suivante:

$$E = -J \sum_{<i,j>} S_i S_j \tag{6}$$

où les variables d'Ising prennent les valeurs 1 ou -1, $< i, j >$ signifiant que la sommation est prise sur les 4 plus proches voisins. J est une constante positive (intégrale d'échange). On considèrera un réseau carré composé de N^2 sites avec les notations suivantes:



avec des conditions périodiques aux bords (comme indiqué sur la figure).

1. La fonction de partition du système étant donnée par:

$$Z(\beta) = \sum_{S_1 S_2 ... S_N} \exp -\beta E \tag{7}$$

où β représente 1/kT et S_i= 1 ou -1. Ecrire la forme analytique de l'énergie moyenne: $< E >= -\frac{\partial Z}{\partial \beta}/Z$ et de la chaleur spécifique $c_v = \frac{\partial <E>}{\partial T}$ (prendre k=1).

2. Dans le cas à 4 sites (N=2), écrire un programme qui calcule $< E >$ et c_v en sommant explicitement sur toutes les configurations de spin possibles. Dans le cas J=1, verifier la présence d'un maximum pour c_v en fonction de T (faites varier T entre 1 et 3) lié à l'existence d'une transition de phase pour ce système.

3. Réfléchir au problème de la programmation pour une valeur de N arbitraire (pas évident, exposer le problème, proposer et programmer éventuellement une solution) et au temps de calcul associé (quelle est la dépendance en N?).

4. Mettre en oeuvre l'algorithme de Metropolis pour un système de taille arbitraire N. Pour cela, écriver un programme qui "visite" les sites les uns après les autres, tirer au hasard la valeur -1 ou 1 (avec générateur de nombres aléatoires fourni) pour le spin courant puis accepter ou rejeter avec le critère de Metropolis (et faire cela un grand nombre de fois). Dans le cas N=2, vérifier que vous retrouvez les résultats exacts obtenus à la question 2.

5. Dans tout programme Monte Carlo il est IMPERATIF de calculer une estimation de l'erreur sur les moyennes des quantités calculées (estimation des fluctuations statistiques). Soit Q la quantité calculée (ici $Q =< E >, c_v$), estimer l'erreur δQ en utilisant la formule suivante:

$$\delta Q = \frac{1}{\sqrt{M}} \frac{1}{\sqrt{M-1}} \sqrt{\sum_{i=1}^{M} (Q_i - < Q >)^2}$$

où M représente M calculs indépendants donnant Q_i et $< Q >$ est la moyenne des Q_i pour ces M calculs (prendre M=10).
6. Augmenter N et étudier l'évolution des différentes quantités avec la taille.

Remarque importante: Il est fortement conseillé d'introduire un tableau de connection entre sites afin de faciliter la programmation de ce problème. Pour cela, on introduit un tableau nvois(i,k) (i=1,N^2 k=1,4) avec la signification suivante: nvois(i,1) donne le numéro du site de gauche pour le site numéro i, nvois(i,2) le numéro du site d'en haut, nvois(i,3) de droite et nvois(i,4) d'en bas.

De cette facon les sommes doubles sur i,j peuvent être faites simplement en sommant sur les sites i et l'indice k associé.

V. AU VOISINAGE DE LA TRANSITION DE PHASE: L'ALGORITHME DE SWENDSEN-WANG

A. Algorithme de Swendsen-Wang

L'algorithme de Metropolis utilisé dans la section précédente repose sur un mouvement d'essai local: la nouvelle configuration d'essai est construite en tentant de modifier la valeur du spin en un seul site. Or, on sait qu'il existe une température critique dans ce système correspondant à une transition de phase du second-ordre. Au point critique, la longueur de correlation, ξ, définie par la relation suivante:

$$< (S_i - < S_i >)(S_j - < S_j >) > \sim \exp(-\frac{\mid i-j \mid}{\xi}) \quad \mid i-j \mid \text{ grand}$$

où $< ... >$ désigne la valeur moyenne thermodynamique correspondant à l'ensemble canonique, devient infinie. Physiquement, cela signifie qu'il existe des corrélations à très longue portée dans le système et que des agrégats de spins identiques et de tailles arbitraires se forment. Il est clair qu'au point critique les valeurs moyennes obtenues à partir d'un algorithme qui visite l'espace des états par des modifications locales est voué à l'échec. Ce phénomène est connu sous le nom de "ralentissement critique" ("critical slowing down").

Ce qu'il faut, c'est imaginer des mouvements d'essai qui incorporent cet élément physique fondamental du système dans son régime critique. Nous allons présenter une telle solution. Il s'agit d'une variante du travail original de Swendsen et Wang (R.H. Swendsen and J.S. Wang, Phys.Rev.Lett. **58**, 86 (1987)).

Une étape élémentaire de cet algorithme correspond à la construction d'un agrégat (ou cluster) de N_c spins identiques dans le système ($1 \leq N_c \leq N$, N étant le nombre de sites total):

- Tirage d'un site quelconque du système. Ce site noté i est le premier site du cluster en cours de construction

- Les sites voisins notés j de ce site i sont incorporés au cluster avec probabilité: $p(S_i, S_j) = 1 - \exp\left[\mathrm{Min}(0, -\beta J S_i S_j)\right]$ (on dit aussi que la liaison (i,j) est activée avec probabilité p). Plus simplement: si le spin du site j voisin de i a une valeur différente de celui de i, le site j n'est pas ajouté au cluster. Dans le cas contraire, le site j est ajouté avec probabilité $1 - \exp(-\beta J)$.

- Cette procédure d'agrégation de sites au cluster est itérée. Les liaisons avec les voisins extérieurs au cluster de chacun des sites ajoutés sont activées ou non avec la probabilté définie précédemment. La procédure est itérée jusqu'à ce que le cluster se termine (ceci est assuré puisqu'évidemment la taille du cluster ne peut dépasser le nombre de sites).

- Finalement, tous les spins du cluster de N_c sites ainsi formé sont retournés (on dit qu'on fait un flip, c'est à dire, $S_i \rightarrow -S_i$ pour $i \in \mathcal{C}$).

Le premier point important à remarquer est qu'un tel mouvement élémentaire dans l'espace de configuration correspond à un processus ergodique. En effet, notons $S=(S_1, ..., S_N)$ et $S'=(S'_1, ..., S'_N)$ deux configurations quelconques de l'espace de configuration. Il est clair que lors d'un étape élémentaire il existe une probabilité non nulle de construire un cluster de taille unité composé d'un site quelconque du système. On a donc une probabilité non nulle après au plus N pas de connecter deux configurations quelconques de l'espace de phase.

Notre deuxième point va consister à démontrer qu'un tel mouvement d'essai vérifie le bilan détaillé. Tout d'abord explicitons en terme de probabilités le mouvement d'essai proposé. On cherche à calculer la probabilté de transition $P^*(S \rightarrow S')$ connectant deux configuration quelconques. Il est clair que cette probabilité est non nulle que dans le cas où S et S' ne diffèrent que par un flip des spins sur un cluster $\mathcal{C}$. Pour une configuration donnée S, il existe un ensemble fini de clusters susceptibles d'être retournés (N clusters à un site, un certain nombre de clusters de taille 2, etc...), je noterai cet ensemble $\mathcal{C}_S$(l'indice S pour rappeler que cet ensemble dépend de la configuration S). Il est facile de se convaincre que la probabilité de transition d'essai s'écrit:

$$P^*(S \rightarrow S') = \frac{\prod\limits_{<i,j>\in\mathcal{C}} p(S_i, S_j) \prod\limits_{<i,j>\in\partial\mathcal{C}} (1 - p(S_i, S_j))}{\sum\limits_{\mathcal{C}\in\mathcal{C}_S} \prod\limits_{<i,j>\in\mathcal{C}} p(S_i, S_j) \prod\limits_{<i,j>\in\partial\mathcal{C}} (1 - p(S_i, S_j))}$$

La notation $< i, j > \in \mathcal{C}$ signifiant que la liaison (i,j) appartient au cluster $\mathcal{C}$ différenciant les deux configurations S et S', et $< i, j > \in \partial\mathcal{C}$ signifiant qu'il s'agit d'une liaison du bord de cluster qui connecte un site i du cluster à un site j qui n'appartient pas au cluster. Montrons

maintenant que cette probabilité de transition d'essai vérifie le bilan détaillé. Pour cela, il faut calculer le rapport:

$$\frac{P^*(S \to S')}{P^*(S' \to S)} = \frac{\prod\limits_{<i,j>\in\mathcal{C}} p(S_i, S_j) \prod\limits_{<i,j>\in\partial\mathcal{C}} (1 - p(S_i, S_j))}{\prod\limits_{<i,j>\in\mathcal{C}} p(-S_i, -S_j) \prod\limits_{<i,j>\in\partial\mathcal{C}} (1 - p(-S_i, S_j))}$$

Ceci étant vrai parce que les normalisations sont identiques dans les deux cas. Après simplification, on obtient:

$$\frac{P^*(S \to S')}{P^*(S' \to S)} = \prod_{<i,j>\in\partial\mathcal{C}} \left(\frac{1 - p(S_i, S_j))}{1 - p(-S_i, S_j)} \right)$$

$$= \exp\,(\sum_{<i,j>\in\partial\mathcal{C}} \mathrm{Min}(0, -2\beta \mathrm{J} S_i S_j) - \mathrm{Min}(0, 2\beta \mathrm{J} S_i S_j))$$

$$= \exp\,(\sum_{<i,j>\in\partial\mathcal{C}} -2\beta \mathrm{J} S_i S_j)$$

la dernière égalité résultant de la relation:

$$\mathrm{Min}(0, X) - \mathrm{Min}(0, -X) = X$$

Par ailleurs, on a:

$$\frac{P(S')}{P(S)} = \frac{\exp\,(-\beta \mathrm{J} \sum\limits_{<i,j>} S'_i S'_j)}{\exp\,(-\beta \mathrm{J} \sum\limits_{<i,j>} S_i S_j)}$$

$$= \exp\,(\beta \mathrm{J} \sum_{<i,j>\in\partial\mathcal{C}} S'_i S_j - \beta \mathrm{J} \sum_{<i,j>\in\partial\mathcal{C}} S_i S_j)$$

$$= \exp\,(\sum_{<i,j>\in\partial\mathcal{C}} -2\beta \mathrm{J} S_i S_j)$$

Le bilan détaillé est donc vérifié.

B. Simulation du modèle d'Ising au voisinage de la transition

1. En utilisant le programme basé sur l'algorithme de Metropolis local, observez le phénomène de ralentissement critique. Pour cela, on regardera la convergence en fonction du nombre de pas Monte Carlo de l'estimateur de l'énergie moyenne et de la chaleur spécifique au point critique.

2. Mettre en oeuvre l'algorithme de Swendsen-Wang. On programmera les estimateurs

de l'énergie moyenne, de la chaleur spécifique, ainsi que de la taille moyenne des agrégats formés à une température donnée.

3. Analyser la dépendance de la taille des agrégats en fonction de la température.

4. Au point critique, comparer les convergences en fonction du temps calcul obtenue avec l'algorithme local et l'algorithme de cluster.

VI. MÉTHODES DE DIAGONALISATIONS EXACTES ET MÉTHODES DE PROJECTION. L'OSCILLATEUR ANHARMONIQUE QUANTIQUE

A. Théorie

Le problème de la diagonalisation de matrices est présent dans la plupart des simulations des systèmes quantiques. On sait que l'état d'un système quantique est décrit par un vecteur d'un espace vectoriel (espace des états). Dans le cas d'un état pur ce vecteur est une combinaison d'états propres de l'opérateur Hamiltonien. Dans le cas d'un mélange statistique (mécanique statistique quantique) les valeurs moyennes des grandeurs physiques sont représentées par un mélange pondéré des valeurs moyennes associées aux états propres du système. La détermination des états propres de l'opérateur Hamiltonien est donc au centre de toute étude quantique. Il est important de souligner qu'en dernière analyse, la mécanique quantique d'un système complexe peut toujours se ramener à la résolution d'un problème d'algèbre linéaire élémentaire dans un espace vectoriel de dimension, N, finie, la seule difficulté pratique étant bien sûr d'évaluer la limite $N \to \infty$.

Soit M_{ij} une matrice NxN. On cherche à résoudre le problème aux valeurs propres suivant:

$$Mu^{(k)} = \lambda_k u^{(k)} \quad k = 1, N$$

On va supposer, par simplicité, que les éléments de matrice M_{ij} sont réels (généralement, c'est le cas dans les applications). On se placera aussi dans le cas où la matrice est symétrique (cas physique standard pour une matrice représentant une observable physique). La matrice M est donc hermitique (ou auto-adjointe) et toutes ses valeurs propres sont réelles.

Il existe de nombreuses méthodes pour diagonaliser une matrice. Elles dépendent du type de matrice (réelle-symétrique, réelle-asymétrique, tridiagonale, matrice avec une structure de bande (un petit nombre d'éléments non nuls au dessus et en dessous de la diagonale principale), matrice pleine, etc...) et du fait que vous vouliez ou non tout ou partie des valeurs propres et/ou des vecteurs propres.

On entre ici très rapidement dans une situation critique d'un point de vue de la simulation numérique. Jusqu'à maintenant (dynamique moléculaire, Monte Carlo) la quantité d'information à stocker en mémoire centrale ne posait aucun problème. Par exemple, dans le cas de la dynamique moléculaire, si on note N le nombre de particules et D la dimension de l'espace, il faut stocker 2ND coordonnées, c'est à dire qu'il faut essentiellement stocker

2ND mots en mémoire centrale. En général, l'essentiel de la physique correspondant à la limite thermodynamique (N$\rightarrow \infty$) est atteinte pour des valeurs de N telles que:

$$\text{2ND} << \text{taille de la mémoire centrale}$$

Dans le cas de la dynamique moléculaire, l'effort numérique porte essentiellement sur le temps d'exécution qui est proportionnel au nombre de pas élémentaires effectués:

$$\mathrm{T_{CPU}} \sim \mathrm{N_{pas}}$$

Dans le cas des problèmes de diagonalisations exactes, la limite de la possibilité de stockage de la machine est très vite atteinte. Typiquement, deux situations se présentent:

- Méthodes de diagonalisations exactes: calcul complet de toutes les valeurs propres et vecteurs propres de la matrice. Effort numérique:

$$\begin{cases} \text{Mémoire centrale} \sim N^2 \\ \mathrm{T}_{CPU} \sim N^3 \end{cases}$$

- Méthodes de projection (Lanczòs, Davidson, Olsen, etc...): calcul de un ou quelques états proches de l'état fondamental. Effort numérique:

$$\begin{cases} \text{Mémoire centrale} \sim 3N \text{ (pour des matrices creuses, hamiltoniens locaux)} \\ \mathrm{T}_{CPU} \sim 100N \end{cases}$$

Illustrons ces deux types d'approche sur un problème quantique simple, quoique non soluble.

B. Oscillateur anharmonique quantique

On va se proposer de résoudre le problème d'une particule dans un potentiel anharmonique à une dimension (ressort anharmonique quantique). Il s'agit de résoudre le problème suivant:

$$H\phi_i = E_i\phi_i$$

avec

$$H = -\frac{1}{2}\frac{d^2}{dx^2} + \frac{1}{2}x^2 + gx^4$$

Pour cela, on écrit les solutions sur une base complète discrète de fonctions élémentaires

$$\phi_k(x) = \lim_{N\to\infty} \sum_{i=1}^{N} c_{ki}\psi_i(x)$$

où l'ensemble des fonctions ψ_n est solution du problème au valeurs propres de l'oscillateur harmonique et forme donc un ensemble complet:

$$\psi_n(x) = (\frac{1}{\pi})^{1/4}\frac{1}{\sqrt{2^n n!}}H_n(x)e^{-\frac{x^2}{2}}$$

où $H_n(x)$ sont les polynômes d'Hermite.

Pour résoudre notre problème il faut trouver le spectre de la matrice $\mathrm{H}_{ij} =< \phi_i \mid H \mid \phi_j >$

1. Diagonalisation exacte

1. En utilisant le formalisme des opérateurs de création et d'annihilation, calculer H_{ij}. On rappelle les définitions suivantes. Opérateur de création: $a^+ = \frac{1}{\sqrt{2}}(x - \frac{\partial}{\partial x})$, opérateur d'annihilation: $a = \frac{1}{\sqrt{2}}(x + \frac{\partial}{\partial x})$, avec commutateur $[a, a^+] = 1$. On peut montrer assez simplement que:

$$\psi_n(x) = \frac{1}{\sqrt{n!}} {a^+}^n \psi_0(x)$$

Montrer que:

$$x\psi_n(x) = \frac{1}{\sqrt{2}}(\sqrt{n}\psi_{n-1}(x) + \sqrt{n+1}\psi_{n+1}(x))$$

et, en itérant cette relation, trouver l'expression explicite des éléments de matrice de H.
2. En utilisant un programme de diagonalisation que l'on vous fournira (par exemple, programme JACOBI), calculer les valeurs propres et les vecteurs propres de la matrice obtenue en tronquant la base à une valeur maximum N. Vérifier que les niveaux d'énergie décroissent en fonction de la dimension de la base utilisée et convergent à partir d'une certaine taille. Déterminer la dépendance de l'effort numérique en fonction de N.
3. Tracer la fonction d'onde fondamentale pour différentes valeurs de g en utilisant les relations suivantes pour évaluer les polynômes d'Hermite:

$$H_0 = 1 \qquad H_1 = 2x \qquad H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x) \tag{8}$$

Commenter les résultats.

2. Méthodes de projection (méthode des puissances, méthode de Lanczòs)

1. Méthode des puissances (Power method)

Nous voulons déterminer l'état fondamental du système. Toute matrice réelle symétrique peut s'écrire sous la forme (décomposition spectrale):

$$H = \sum_k \lambda_k \mid \phi_k >< \phi_k \mid$$

Dans ce qui suit, les valeurs propres sont classées par ordre croissant. On peut donc écrire:

$$(E - H)^n = \sum_k (E - \lambda_k)^n \mid \phi_k >< \phi_k \mid$$

La méthode des puissances repose sur le résultat évident que:

$$\lim_{n\to\infty} (E - H)^n \sim \mid \phi_0 >< \phi_0 \mid$$

sous la condition que: $\mid \frac{E-\lambda_0}{E-\lambda_k} \mid < 1$ pour tout k, ce qui est vrai si la constante E est choisie, par exemple, plus grande que toutes les valeurs propres. Notons $\mid \Psi >$ un vecteur quelconque non orthogonal au vecteur propre fondamental, on peut donc extraire le vecteur

correspondant à l'état fondamental par application successive de la matrice (E-H).

Application au problème de l'oscillateur anharmonique:

a. Mettre en oeuvre l'algorithme qui permet de calculer $\mid \Psi^{(n)} > \equiv (E-H)^n \mid \Psi >$ où $\mid \Psi >$ est un vecteur quelconque initial.
b. En prenant comme vecteur initial un ensemble de coefficients tirés au hasard, vérifier que le vecteur $\mid \Psi^{(n)} >$ converge vers l'état fondamental. Comparer avec les résultats de la méthode de diagonalisation exacte.

2. Méthode de Lanczòs

Cette méthode généralise la méthode précédente. L'idée fondamentale est très simple. Plutôt que de prendre comme vecteur d'essai pour H le vecteur $\mid \Psi^{(n)} >= (E-H)^n \mid \Psi >$, la méthode consiste à prendre comme vecteur d'essai le meilleur vecteur construit à l'aide de l'ensemble de tous les vecteurs $\mid \Psi^{(k)} >$ k=0,n. Dit autrement, la méthode consiste à diagonaliser H dans la base: $\{\mid \Psi >, H \mid \Psi >, H^2 \mid \Psi >, ..., H^n \mid \Psi >\}$. L'espace vectoriel sous-tendu par cet ensemble est quelque fois appelé l'espace de Krylov associé à Ψ et H. On va voir que la convergence dans un tel espace est énormément augmentée.

D'un point de vue pratique, il est souhaitable de disposer d'une base orthonormale, on va donc effectuer une orthonormalisation systématique des vecteurs construits.

Partant de $\mid \Psi >$ notre vecteur arbitraire de depart, on construit le premier vecteur de Lanczòs normalisé:

$$\mid u_0 >= \frac{\mid \Psi >}{\sqrt{< \Psi \mid \Psi >}}$$

L'étape suivante consiste à calculer H$\mid u_0 >$ On construit ensuite le second vecteur de Lanczòs $\mid u_1 >$ à l'aide de ce vecteur et du vecteur de Lanczòs précédent $\mid u_0 >$:

$$\beta_2 \mid u_1 >= H \mid u_0 > -\alpha_1 \mid u_0 >$$

α_1 est déterminé en imposant que $\mid u_1 >$ soit orthogonal à $\mid u_0 >$ et β_2 déterminé pour que le vecteur $\mid u_1 >$ soit de norme unité:

$$\alpha_1 =< u_0|H|u_0 >$$

$$\beta_2 = \sqrt{< u_0|H^2|u_0 > - < u_0|H|u_0 >^2}$$

A l'itération suivante on écrit le nouveau vecteur sous la forme:

$$\beta_3 \mid u_2 >= H \mid u_1 > -\alpha_2 \mid u_1 > -\beta_2 \mid u_0 >$$

Noter que, par construction, le nouveau vecteur est orthogonal à $\mid u_0 >$. α_2 est obtenu en imposant l'orthogonalité à $\mid u_1 >$ et β_3 pour que le vecteur soit de norme unité:

$$\alpha_3 =< u_1|H|u_1 >$$

Finalement, on arrive au cas général:

$$\beta_{i+1} \mid u_i >= H \mid u_{i-1} > -\alpha_i \mid u_{i-1} > -\beta_i \mid u_{i-2} > \quad i = 1, ..., N$$

avec:

$$\alpha_i = < u_{i-1}|H|u_{i-1} >$$

et

$$\beta_{i+1} = \sqrt{< u_{i-1}|H|u_{i-1} >^2 - \alpha_i^2 - \beta_i^2}$$

La matrice représentant H dans cette base est donc tridiagonale:

$$\begin{pmatrix}
\alpha_1 & \beta_2 & 0 & . & . & . & . & . & . & . & 0 \\
\beta_2 & \alpha_2 & \beta_3 & 0 & . & . & . & . & . & . & . \\
0 & \beta_3 & \alpha_3 & \beta_4 & 0 & . & . & . & . & . & . \\
. & . & . & . & . & 0 & . & . & . & . & . \\
. & . & . & . & . & \beta_i & 0 & . & . & . & . \\
. & . & . & . & . & \alpha_i & . & . & . & . & . \\
. & . & . & . & 0 & \beta_{i+1} & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & 0 \\
. & . & . & . & . & . & . & . & . & . & \beta_n \\
. & . & . & . & . & . & . & . & 0 & \beta_n & \alpha_n
\end{pmatrix}$$

et il existe des méthodes performantes pour diagonaliser les matrices tridiagonales (algorithme QL, par exemple, voir Numerical Recipes).

Faisons un certain nombres de remarques concernant cet algorithme. *A priori*, si on s'intéresse essentiellement à la détermination de l'état fondamental le vecteur initial $|\Psi >$ sera choisi le plus proche possible de l'état fondamental recherché. A la première itération (n=1), l'énergie obtenue sera l'énergie variationnelle associée au vecteur d'essai, c'est à dire $E_v = \frac{<\Psi|H|\Psi>}{<\Psi|\Psi>}$. Cette énergie peut donc être déja une très bonne approximation de l'énergie exacte. A la seconde itération (n=2), le vecteur $|u_1 >$ est à un facteur près le vecteur: $H|\Psi > -E_v|\Psi >$. Si $|\Psi >$ est le vecteur exact $|u_1 >$ est nul, $\beta = 0$ et le processus est arrêté: on a construit un espace qui contient la solution exacte (l'espace vectoriel est stable ou fermé sous l'action de H). Dans le cas contraire, $|u_1 >$ représente la direction privilégiée pour calculer le résidu entre l'état variationnel et l'état exact. En continuant ainsi on se rapproche très vite de la solution exacte. En itérant, β va essentiellement devenir de plus en plus petit. Lorsque β s'annule cela signifie que H appliqué au dernier vecteur Lanczòs s'écrit dans la base de tous les vecteurs Lanczòs précédents, on peut donc arrêter le processus itératif, l'espace de Krylov obtenu est fermé.

La convergence en fonction du nombre d'itérations est au moins exponentiellement rapide. Un argument heuristique peut nous en convaincre. A une itération n , l'état fondamental obtenu peut s'écrire sous la forme:

$$\Psi_0 = \sum_{i=0}^{n-1} c_i H^i |\Psi >$$

L'énergie associée à cette fonction d'onde est optimale (principe variationnel) par rapport au coefficient c_i. On a donc:

$$E(\Psi_0) < E(\tilde{\Psi})$$

où:

$$\tilde{\Psi} = \sum_{i=0}^{n-1} \frac{(-1)^i}{i!} H^i |\Psi >$$

Pour n assez grand, on voit donc que $E(\Psi_0)$ est majorée par une quantité qui converge exponentiellement vite vers l'énergie de l'état fondamental.

Problème:

1. En partant du même vecteur initial que précédemment, construire la matrice tridiagonale de Lanczòs.
2. Diagonaliser cette matrice avec la méthode de Jacobi. Etudier la convergence des énergies en fonction de n et discuter l'apparition d'instabilité associées à ce qu'on appelle souvent des solutions "fantômes". On fera cette étude en simple, double, et quadruple précision.

VII. MODÈLE D'HEISENBERG ANTIFERROMAGNÉTIQUE 2D

L'Hamiltonien est défini de la manière suivante:

$$H = \sum_{<i,j>} \mathrm{J}\vec{\mathrm{S}}_\mathrm{i}\vec{\mathrm{S}}_\mathrm{j}$$

sur un réseau bidimensionnel carré avec conditions aux limites périodiques (identique au réseau utilisé précédemment pour le modèle d'Ising 2D). La constante d'échange J est positive. On a:

$$S^l = \frac{1}{2}\sigma_l \quad l = x, y, \text{ ou } z$$

avec les trois matrices de Pauli:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \sigma_y = \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix} \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Définissons les opérateurs suivants:

$$S^+ \equiv S^x + iS^y = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

$$S^- \equiv S^x - iS^y = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

On vérifie facilement leurs propriétés fondamentales:

$$S^+|\uparrow> = 0 \quad S^-|\uparrow> = |\downarrow> \quad S^+|\downarrow> = |\uparrow> \quad S^-|\downarrow> = 0$$

où $|\uparrow> = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ et $|\downarrow> = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Une contribution élémentaire pour un lien peut se réécrire:

$$\vec{\mathrm{S}}_\mathrm{i}\vec{\mathrm{S}}_\mathrm{j} = \frac{1}{2}(S_i^+ S_j^- + S_i^- S_j^+) + S_i^z S_j^z$$

c'est à dire:

$$\vec{S}_i\vec{S}_j|\uparrow\uparrow>=\frac{1}{4}|\uparrow\uparrow>$$

$$\vec{S}_i\vec{S}_j|\downarrow\downarrow>=\frac{1}{4}|\downarrow\downarrow>$$

$$\vec{S}_i\vec{S}_j|\uparrow\downarrow>=\frac{1}{2}|\downarrow\uparrow>-\frac{1}{4}|\uparrow\downarrow>$$

$$\vec{S}_i\vec{S}_j|\uparrow\downarrow>=\frac{1}{2}|\uparrow\downarrow>-\frac{1}{4}|\downarrow\uparrow>$$

On se placera dans la base de dimension 2^N définie par:

$$|m_1.....m_N> \quad \text{avec } m_i=\uparrow \text{ ou } m_i=\downarrow$$

On va mettre en oeuvre un programme pour évaluer l'énergie du fondamental ainsi qu'un certain nombre de ses propriétés. Il est important de remarquer que de sévères limitations de mémoire centrale apparaissent très vite dans ce type de problème (on a 2^N états ! à considérer). Il faut donc faire très attention à ne pas utiliser des tableaux inutilement grands, c'est l'objet de la première question.

1. L'étape fondamental de toute méthode de projection est la partie qui calcule l'application de la matrice Hamiltonienne sur un vecteur donné de l'espace vectoriel. Construire les trois tableaux suivants qu'on pourra appeler, par exemple, h(i,j), nhstate(i), et nconnect(i,j) avec la signification suivante:

$$H|i>=\sum_{j=1}^{nhstate(i)} h(i,j)|nconnect(i,j)>$$

où $|i>$ représente un élément quelconque de la base. "nhstate(i)" représente le nombre total d'états de la base connectés à l'état $|i>$, "nconnect(i,j)" donne le numéro du jième état connecté à $|i>$, et "h(i,j)" est l'élément de matrice non nul correspondant. De cette manière, seuls les éléments de matrice non nuls de la matrice Hamiltonienne sont stockés. Soit $|\Psi>$ un vecteur arbitraire de composantes c_i dans la base, on a:

$$H|\Psi>=\sum_{i=1}^{nsttot} c_i \sum_{j=1}^{nhstate(i)} h(i,j)|nconnect(i,j)>$$

où nsttot représente la dimension de la base. Afin de faciliter la programmation on pourra représenter un état quelconque de la base à l'aide d'un entier en base 2:

$$M(m_1,...,M_N)=n_1 2^0+n_2 2^1+....+n_N 2^{N-1}$$

où n_i =0 si par exemple le site i est occupé par un spin down et n_i=1 si le site est occupé par un site up. Ceci permet de boucler sur tous les états en parcourant les entiers de 0 à $(2^{N-1}-1)$. A partir d'un entier donné on peut remonter à l'occupation des sites très

simplement en utilisant les fonctions de manipulation de bits du Fortran. On a les trois fonctions élémentaires:

$$\text{logic} = \text{btest(M, i)}$$

logic vaut .true. si le ième bit de l'entier M vaut 1, sinon logic =.false. Sur la machine que vous utilisez, les entiers sont par défaut représentés sur 32 bits et le bit 0 est le bit de poids faible (bit 1 dans la représentation précédente).

$$\text{M}' = \text{ibset(M, i)}$$

Le ième bit de M est mis à 1 et l'entier résultant est M'

$$\text{M}' = \text{ibclr(M, i)}$$

Le ième bit de M est mis à zéro.

2. Appliquer la méthode de puissance pour calculer l'état fondamental du système. Regarder la dépendance en fonction du nombre de sites. Vérifier que l'état fondamental est un singulet, c'est à dire est état propre de S^2 :

$$S^2 = (\sum_{i=1}^{N} \vec{S}_i)^2$$

Evaluer également l'aimantation alternée définie par:

$$M(N) = < 0 | (\sum_{i \in A} S_i^z - \sum_{i \in B} S_i^z)^2 | 0 >$$

où $|0>$ dénote l'état fondamental. Discuter les résultats.

3. Calculer l'état fondamental par méthode de Lanczòs.

4. Calcul de la fonction de Green de spin. Montrer que la transformée de Fourier de la fonction de Green peut s'exprimer comme une valeur moyenne de l'inverse de H. Exprimer cette fonction de Green sous forme de fraction continue. Discussion physique des résultats.